ISI Reprint Series
ISI/RS-89-231
February 1989

(4)

University of Southern California

Pedro Szekely

Structuring Programs to Support Intelligent Interfaces

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA206284

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RS-89-231	5. MONITORING ORGANIZATION REPORT NUMBER(S) -------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION -------
6c. ADDRESS (City, State, and ZIP Code) 4676 Admiralty Way Marina del Rey, CA 90292-6695	7b. ADDRESS (City, State, and ZIP Code) -------
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA Air Force Logistics Command 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-87-C-1499 F33600-87-C-7047 MDA903-86-C-0178
8c. ADDRESS (City, State, and ZIP Code) DARPA 1400 Wilson Boulevard Arlington, VA 22209 Air Force Logistics Command Wright-Patterson Air Force Base Ohio 45433-5320	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. ------- PROJECT NO. ------- TASK NO. ------- WORK UNIT ACCESSION NO. -------

11. TITLE (Include Security Classification)
Structuring Programs to Support Intelligent Interfaces (Unclassified)

12. PERSONAL AUTHOR(S) Szekely, Pedro

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Research Report	FROM _____ TO _____	1989, February	27

16. SUPPLEMENTARY NOTATION
This report is a revised version of a paper that appeared in *Architectures for Intelligent Interfaces: Elements and Prototypes*, Monterey, California, March 1988.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
intelligent interfaces, UIMS, user interface management systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The ability to connect user interface building blocks with a wide variety of application programs is crucial in the construction of intelligent interfaces. This paper presents a language to specify the communication between building blocks and application programs, with two important features: First, the language is abstract enough to isolate the application program from the details of particular interface styles. Second, the language is rich enough to support the communication of the information needed for the low-level aspects of the user interface. The paper also describes a UIMS based on this language, and discusses how the language supports tools that can reason about the building blocks provided by the UIMS.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Victor Brown Sheila Coyazo	22b. TELEPHONE (Include Area Code) 213/822-1511
	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ISI Reprint Series
ISI/RS-89-231
February 1989

University of Southern California

Pedro Szekely

Structuring Programs to Support Intelligent Interfaces

INFORMATION SCIENCES INSTITUTE

213/822-1511

4676 Admiralty Way/Marina del Rey/California 90292-6695

The initial research was supported by the Defense Advanced Research Projects Agency, ARPA Order No. 4976, Amendment 20, under Contract No. F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB. Further work has been supported in part by DARPA under Contract No. MDA903-86-C-0178 and in part by the Air Force Logistics Command under Contract No. F33600-87-C-7047. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Air Force Logistics Command, the U.S. Government, or any person or agency connected with them.

1 Introduction

Many user interface management systems (UIMSs) have a library of interface building blocks that can be used to assemble the interface for a program [Tanner83]. An *intelligent* UIMS has a reasoning component that determines how to use this library based on a model of the program, the user and interface design. For instance, the Integrated Interfaces system [Arens88] uses a model of the objects and operations in the program, and a model of interface building blocks such as icons, menus, and maps. The reasoning component uses a set of rules to choose what interface building blocks to use, and then connects them with the objects and operations of the application.

Connecting the interface building blocks with different application requires that the building blocks use a standard language to communicate with the application objects and operations. Otherwise, it is necessary to write a program to translate between the languages used by the building blocks and the application, making it impossible to automatically connect building blocks and applications.

Suppose, for instance, that applications are programmed using an object-oriented programming language. The application objects would be defined as classes, and the application operations would be defined as methods for those classes. In such an environment, the standard language for user interface/application communication would consist of a list of methods that each application object must provide. If each application object provides these methods, then the interface building blocks could communicate with any application object since the building blocks would use only these methods to communicate with the application. Such a standard language would make the building blocks plug-compatible with a large number of application programs.

Defining such a standard language is difficult because the language must support a wide variety of interface styles and a wide variety of applications programs. This generality requirement generates two conflicting goals:

- The language must define the communication with the application in abstract terms, without referring to any particular interface style. Otherwise, the application will not be independent of interface style, and hence cannot support a variety of interfaces.

- The language must support the communication of the information needed for each interface style, including the information needed to support application-dependent feedback (semantic feedback). Since different interaction styles provide different kinds of feedback (*e.g.* rubberbanding, gravity, highlighting), it appears that the language must be tuned to particular interface styles.

This paper describes a language that achieves a good compromise between these two conflicting goals. To achieve the generality goal, the language is based on an extension of the *language view* of programs [Foley82, Moran81, Newman79], a general model of interactive programs. The extensions involve mainly enriching the description of the semantics of a program. To ensure that the language supports a wide variety of interface styles, the language primitives were tested with a large number of interaction styles, and refined to make sure that the appropriate information can be communicated. The resulting language primitives allow the communication of the application-specific information needed in many interface styles, but the information is described at a high enough level of abstraction so that the details of the interface styles remain hidden.

The paper is organized as follows. Section 2 presents the language view of programs. Section 3 introduces the primitives of the proposed language, and section 4 shows how these primitives can explain the behavior of the interface to a chess program. Section 5 describes Nephew, a UIMS based on the proposed communication language, and shows how the interface to the chess program is implemented with Nephew. Finally, section 6 describes how the language can be used in the reasoning component of intelligent interfaces.

2 The Language View of Programs

Communication is "the exchange of meanings between individuals through a common system of symbols" [Encyclopae74]. To communicate a meaning, called a *concept* in this paper, the sender must encode the concept into a sequence of symbols with a concrete physical representation that can be perceived by the receiver. The receiver must decode the symbols it perceives, and extract the corresponding concept.

A human and a program communicate concepts by encoding them as changes in the state of input and output devices attached to the computer. Users communicate

concepts to programs by manipulating the input devices. Programs monitor the state of the devices to recover the concepts, process them, and produce new concepts, which are transmitted to the user by encoding them in the state of the output devices.

The primitives of the proposed language were identified by classifying the concepts that users and programs communicate via graphical user interfaces. Each concept category corresponds to a primitive of the language. Put in object-oriented terms, each concept category corresponds to a method that computes the information embodied in the concept.

The classification of the concepts that users and programs communicate, called *communication concepts*, is based on the *language view* of programs [Foley82, Moran81, Newman79]. According to the language view, the language to communicate with a program has four levels called the *conceptual*, *semantic*, *syntactic* and *lexical* levels. The conceptual level describes the tasks the user is able to accomplish using the program. Tasks are specified by decomposing them into subtasks, and subtasks are decomposed further until they can be accomplished using a single operation provided by the program. The semantic level describes these operations and the objects that they operate on. The syntactic and lexical levels describe how the user accesses the objects and operations using the input and output devices.

These levels are consistent with the definition of communication given at the beginning of this section. The communication concepts are the meanings encoded in the lexical and syntactical levels. The user transmits them in order to use the facilities provided by the program, *i.e.* the objects and operations.

Even though the division into levels appears intuitive, no precise notation for describing the levels of an application program exists. Moran's CLG [Moran81] is perhaps the most complete notation, but it is not precise enough to be executable, and many aspects of graphics-based interfaces cannot be expressed in it. Most of the work on notations for describing programs according to the language model has concentrated on the syntactic and lexical levels. BNF, transition networks, and event based systems are the most popular methods [Green86].

The specification of the semantic level usually consists of a list of type and procedure declarations like those used in MIKE [Olsen86] and the ones used in Foley's *et. al.* user interface design environment [Foley88]. The idea behind such specifications is that the types specify the objects that the application provides, and the procedures

specify actions that can be performed on those objects.

This kind of specification does not capture many attributes of a program that are conveyed to the user via graphical user interfaces. For instance, the user interface for a chess program could highlight the pieces that can be legally moved at any given moment. To highlight the pieces the user interface has to somehow find out which pieces can be moved. Some supporters of the type/procedure method of specification would say that one could simply include in the specification a procedure that computes the relevant information. This is a poor solution because such a specification does not record the purpose of the procedure, which is to test whether a given piece is a legal parameter for the move operation. This specification would make it impossible to construct a generic user interface building block that highlights objects that are correct parameters to an operation. The building block would not know which procedure to call, and even if it knew, it would not know what parameters to provide and in what order. The specification language should capture more about the meaning of the types and procedures. The language should allow one to express that the "test-chess-piece" procedure tests whether an object is a legal candidate to become an operation input. Any interface building block that needs this information can get it by using the appropriate language construct.

Some UIMSs [Hayes85, Smith84] specify the semantics of programs using formalisms that specify more than types and procedures, and capture the information needed to support the generation of menu-based interfaces. Unfortunately, they do not capture some of the aspects of the semantics of a program needed to support highly interactive graphical user interfaces.

The difficulties with the specification of the levels of the language model might suggest that the model is inappropriate [Kamran83]. This paper shows that many of the difficulties of the language model can be overcome by enriching the semantic level description.

Finally, the classification of the concepts that users and programs communicate is also based on an analysis of the user interface of many Macintosh programs (*e.g.* MacDraw, Excel and Finder). The communication concepts can explain the behavior of the interface features of these programs.

The communication concepts introduced in the next section capture the distinctions in semantics that are relevant to the construction of graphical user interfaces.

Each class of communication concept captures a different semantic distinction.

3 Communication Concepts

The communication concepts are divided into *input* and *output* communication concepts. The input communication concepts are the meanings encoded in the gestures produced with the input devices (syntactic and lexical levels). These concepts express what the user wants to do with the objects and operations (semantic level).

The set of concepts that a user might want to express about the objects and operations of a program are open ended. The power of the set of communication concepts presented below is that the set is small, yet it is rich enough to explain the behavior of a large variety of graphical user interfaces. The set of interfaces includes mouse based systems such as the Macintosh interface. The communication concepts can explain the effect of every input event, every mouse movement, for a large variety of interfaces. Section 4 illustrates this using a chess program.

The input communication concepts are used by the user to communicate to a program:

Activate *operation*: expresses the user's intention to invoke an operation. After activation the user performs other activities, which are described below, and then executes the operation.

Execute *operation*: requests that an operation be executed. If the operation is ready to be executed (*e.g.* all required inputs have been specified), the appropriate procedure is called. Otherwise, a communication concept is transmitted to tell the user about the error.

Bind *input* **of** *operation* **to** *value*: specifies a value for an input.

Preview *operation*: computes an approximation of the effects that executing the operation would have, given the current setting of the input values. The rubberbanding effect in many graphical operations is an instance of the use of the **preview** concept.

Abort *operation*: requests that the execution or activation of an operation be terminated.

These communication concepts can be qualified with the qualifiers **plan** and **not-plan**. Where as the **bind** *input* **to** *value* concept specifies the value of an input to an operation, the **plan bind** *input* **to** *value* concept tells the program that the user has entered a state in which he or she can specify the given input. For instance, moving the mouse over a check box icon can be interpreted as a plan to set a value. Clicking the mouse will set the value, but moving the mouse away from the check box without clicking will not set the value, and can be interpreted as a **not-plan bind** *input* concept. Section 4 illustrates how these qualifiers are used in the interface for a chess program.

The output communication concepts are used by the program to communicate to the user:

Contents: the state of an object.

Changes: a change in the contents of an object.

The following set of output communication concepts refers only to operations:

Alternatives: the set of values from which an input for an operation must be chosen.

Correct *input* **of** *operation*: the value **true** if the input of an operation is bound to a valid value; otherwise, a description of why the binding is incorrect.

Using *object* **as** *input* **of** *operation*: a concept that specifies that a given object is being used as an input value for an operation.

More details about the meaning of the communication concepts, and the role they play in the communication of a user with a program can be found in the author's thesis [Szekely88].

These communication concepts are the categories in a classification. To precisely describe a program it is often necessary to define specializations of some of these concepts. For instance, the **changes** communication concept reports that an object has changed, but does not specify how. A specialization of the **changes** concept should be used when it is desired to specify how an object changes.

For example, a chess program could transmit the PIECE *changes* communication concept when PIECE is moved from one location to another, or when PIECE is taken. The **changes** communication concept could be specialized for the chess program into, say, **changes-moved** and **changes-taken** to distinguish between the two kinds of changes.

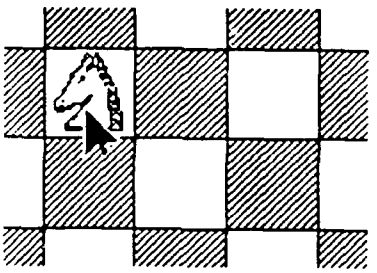
These two concepts would allow the chess program to transmit more accurate information about changes.

The most important point about the communication concepts is that they can express the communication between a user and a program in abstract terms, without reference to particular interaction techniques. The interpretation of every input event, and all display updates can be viewed as the encoding of one of the communication concepts.

4 An Example

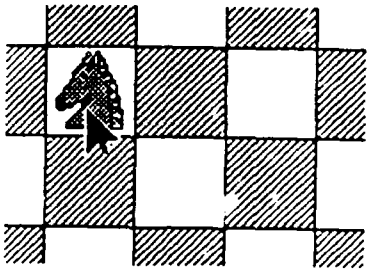
This section illustrates how the communication concepts can explain the input/output behavior of a program. The figures that follow show snapshots of a chess program display while the user drags a piece using the mouse. The interface behaves as follows: when the user presses the mouse button over a piece that can be moved, the piece is highlighted, then an outline of the piece follows the mouse. The user can then drag the piece to its destination and release the mouse button to drop the piece there. While dragging, the location under the mouse highlights when it is a legal destination for the piece.

The following screen snapshot shows the chess board after the user moves the mouse over the knight, but has not pressed the mouse button.

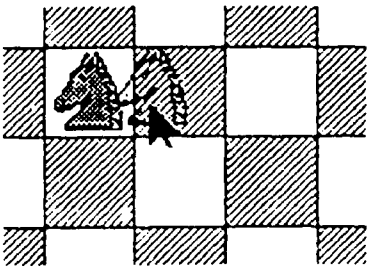


The following snapshots consist of two parts. The first part describes an input event and its related communication concepts, and the second one shows the new screen state after the communication concepts are processed. In what follows the *event* describes the input event received by the program, the *input CC* is the input communication concept encoded in the event, the *output CC* is the output communication concept produced as a result of processing the input CC, and the *display* is a description of how the program displays the output CC.

Event: the user presses the mouse button over the knight.
Input CC: **bind the** PIECE **input of the** MOVE **operation to** PIECE-UNDER-MOUSE.
Output CC: **the** PIECE **input of the** MOVE **operation is correct.**
Display: highlight the piece.

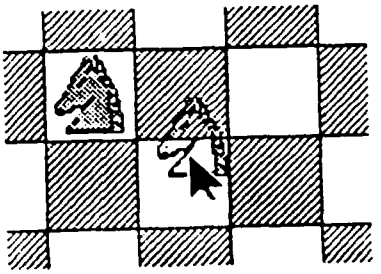


Event: user moves the mouse to adjacent square.
Input CC: **plan to bind the** LOCATION **input of the** MOVE **operation to** LOCATION-UNDER-MOUSE.
Output CC: **the** LOCATION **input of the** MOVE **operation is incorrect.**
Display: **nil**, only correct locations are highlighted.



The program tells the user that if the piece is dropped in that location it is an illegal move. Should the user release the mouse button at this point the program would bind the location input to an incorrect value, beep to present the error, and de-activate the operation.

Event: user moves the mouse to adjacent square.
Input CC: **plan to bind the** LOCATION **input of the** MOVE **operation to** LOCATION-UNDER-MOUSE.
Output CC: **the** LOCATION **input of the** MOVE **operation is incorrect.**
Display: **nil**, only correct locations are highlighted.

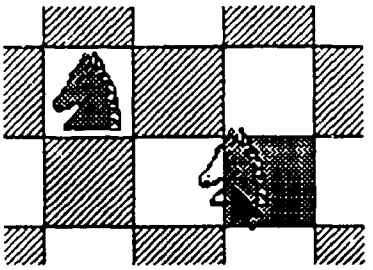


Event: user moves the mouse to adjacent square.

Input CC: **plan to bind the** LOCATION **input of the** MOVE **operation to** LOCATION-UNDER-MOUSE.

Output CC: **the** LOCATION **input of the** MOVE **operation is correct.**

Display: highlight the location.

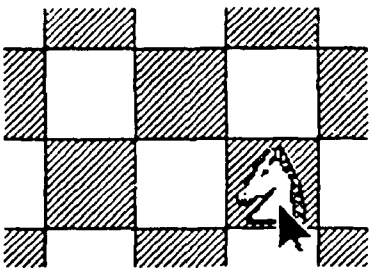


Event: user releases the mouse button.

Input CC: **bind the** LOCATION **input of the** MOVE **operation to** LOCATION-UNDER-MOUSE, **execute the** MOVE **operation.**

Output CC: **changed the** PIECE **input of the** MOVE **operation.**

Display: Erase the piece from the old location and display it at the new location.



Should the user move the mouse to an adjacent location without releasing the mouse button, the program would interpret the event as a **not-plan to bind**, and would remove the highlighting from the location.

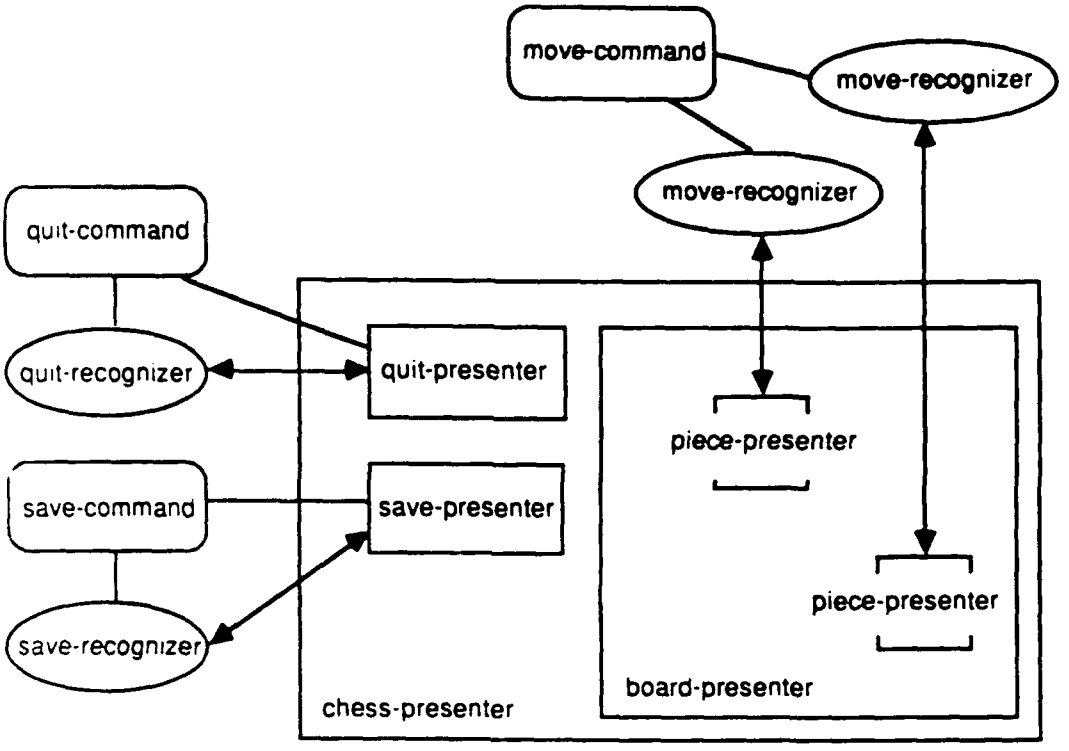


Figure 1: The architecture of the chess program.

Note that even the low level details of the interface such as displaying feedback in response to individual mouse movements can be expressed in terms of the communication concepts.

5 Nephew: A UIMS Based on Communication Concepts

The communication concepts described above serve, not only to explain the behavior of graphical user interfaces, but also as the foundation of a UIMS. This section describes Nephew, a UIMS based on communication concepts [Szekely88]. Nephew is the successor to the COUSIN [Hayes85] UIMS.

Figure 1 shows the architecture of a typical Nephew application, in this case a chess program that behaves as described in section 4. The program consists of four kinds of building blocks, called recognizers, commands, presenters, and the application objects (not shown in the figure).

Recognizers. A recognizer is parser for a complete input gesture such as a mouse

click or a mouse drag. A recognizer produces communication concepts in response to input events in the gesture, and sends the concepts to a command for interpretation.

Commands. A command is a communication concept interpreter for one specific operation. Each command receives communication concepts from recognizers and responds to them producing output communication concepts, which it sends to presenters to be displayed.

Presenters. A presenter displays communication concepts.

As shown in figure 1, an application implemented with Nephew typically consists of many recognizers, commands and presenters. **Quit-command** and **save-command** are commands for the quit and save operations provided by the chess program. They are made accessible to the user by displaying them with the presentation produced by **quit-presenter** and **save-presenter** presenters. The **quit-recognizer** and **save-recognizer** are defined to activate their commands when they detect a click over their presentation. For instance, if the user clicks the mouse over the **quit-presenter**, then **quit-recognizer** sends an **activate** message to **quit-command**. Since the quit operations takes no parameters, **quit-command** responds to **activate** by sending itself an **execute** message, thus initiating the execution of the **quit** operation. The **move-recognizers** and the **move-command** implement the behavior discussed in section 4. Each piece presenter has a **move-recognizer** to handle drag gestures that start at the piece. While dragging a piece, **move-recognizer** sends to **move-command** the sequence of communication concepts listed in section 4. The output communication concepts are displayed by each **piece-presenter** and by **board-presenter**.

Nephew is implemented in Lisp on a Symbolics Lisp Machine using the Flavors object-oriented programming package. Nephew provides predefined building blocks of each kind, implemented as classes, and the job of the interface implementor is to assemble these building blocks to construct a program. The following subsections discuss the implementation in more detail.

Recognizers

A recognizer is a parser for a complete input gesture such as a mouse click or a mouse drag. Nephew implements recognizers as classes, one class for each kind of recognizer.

Recognizer-Basic implements the behavior common to all recognizers. This includes methods to turn recognizers on and off, methods to connect recognizers to presenters, and methods get the input events from a global event queue. The response to the input events is implemented by the subclasses listed below.

Click-Recognizer implements the click gesture. The user interface implementor must supply the communication concepts that the recognizer transmits when the click starts and terminates.

Drag-Recognizer implements the drag gesture. The default Drag-Recognizer is programmed to drag an outline of its presenter while the gesture is in progress. The implementor must provide the communication concepts that the recognizer transmits when the drag gesture begins, each time the mouse moves, and when the gesture terminates.

Dispatch-Recognizer implements the standard dispatch behavior to distribute events to other recognizers. Instances of Dispatch-Recognizer are attached to presenters with sub-presenters to dispatch events to the presenters attached to the sub-presenters.

Window-Move-Mixin is a subclass of Drag-Recognizer that allows windows to be moved by dragging them by their title bar.

For example, in the chess program implementation shown before in figure 1, the **move-recognizers** are instances of Drag-Recognizer, and **quit-recognizer** and **save-recognizer** are instances of Click-Recognizer. Instances of Dispatch-Recognizer, not shown in the figure, are attached to **board-presenter** and **chess-presenter** to distribute the input events to the other recognizers.

Commands

A command is a communication concept interpreter for one specific operation. Nephew implements commands as classes, and implements input communication concepts as

methods of command classes. The following are the methods that implement the communication concepts:

execute: executes the procedure that implements the operation.

preview: invokes the procedure that implements the preview.

activate: makes the command active.

deactivate: makes the command inactive.

cancel: cancels the execution of an operation.

Newly defined command classes inherit default implementations for all of these methods. Implementors can override some of these methods to implement different interface styles.

In addition, a command must provide the following methods for each *input* of an operation. Nephew constructs default implementations for these methods from the operation declarations, but the user interface implementor can override them to provide non-standard behavior:

input: a command must provide a method called *input* corresponding to each operation input called *input*. These methods access the value of inputs or their plan component. For instance, the **move-command** used in the chess program provides methods called **piece** and **location**, corresponding to the piece and location parameters of the move operation.

set-*input*: likewise, a command must provide a method called set-*input* corresponding to each operation input called *input*. These methods set the value of inputs or their plan component. For instance, **move-command** provides methods **set-piece** and **set-location** to set the command's piece and location inputs.

test-*input*: likewise, a command must provide a method called **test-***input* to test whether a value is a legal input.

alternatives-*input*: generates the set of valid inputs for input.

For instance, the **move-command** used in the chess program overrides the **test-piece** and **test-location** methods with predicates that define the rules of chess. **Move-command** also overrides the **alternatives-piece** method to return the set of pieces that can be

moved at any given time, and the **alternatives-location** method to return the locations to which the selected piece can be moved. The alternatives methods are not used in the chess interface described above, but they could be used in an interface that highlights the pieces that can be moved, and the locations where a selected piece can be moved. No changes to the chess application would be needed to add this feature to the user interface.

Nephew also provides a library of command classes that implement common interface styles:

Command-Basic provides the behavior common to all commands by defining default implementations for the methods listed above.

Input-Prompt-Mixin provides facilities to associate a set of recognizers with each command input, and to switch on the appropriate recognizers when the application program requests input. Switching the recognizers on will allow them to receive input events and thus decode communication concepts that set the inputs.

Hour-Glass-Mixin automatically runs the **hour-glass-recognizer** while an operation is executing. This recognizer displays an hour-glass cursor and intercepts all input events.

Confirmation-Mixin overrides the **execute** method. Before invoking the application operation it switches on a recognizer to prompt the user for a confirmation to execute the operation.

For example, in the chess program the **quit-command** is an instance of a command class that includes **Confirmation-Mixin** so that the user is asked to confirm before exiting the chess program. The **save-Command** uses the **Input-Prompt-Mixin** to prompt for the file to save the state of the game.

Presenters

A presenter displays communication concepts. Nephew implements presenters as classes, one class for each particular way of displaying the communication concepts for each kind of object. Nephew provides presenter classes to display a variety of generic application classes such as lists, structures and array, and also to display commands and even presenters themselves.

In Nephew complex presentations are constructed by connecting several presenter instances to form a tree. Presenters with children are called *parent* presenters, and the children are called *sub-presenters*. For instance, in the chess example the board is a presenter, and the pieces are sub-presenters of the board presenter.

The following are the presenter classes in Nephew's library:

Presenter-Basic provides the basic facilities to link presenters to the objects they present, and all the hooks into the graphics package. Presenter-Basic displays its object as a string identifying the type of object (*e.g.* a chess-piece). This is useful in the initial stages of the implementation of a user interface.

Borders-Mixin provides the facilities to define the borders, background and foreground of presenters.

Window-Mixin provides the facilities to attach a presenter to a window.

Structured-Presenter provides the facilities for a presenter to have sub-presenters.

Record-Presenter, List-Presenter are sub-classes of Structured-Presenter that can present records and lists. They provide the methods that know how to construct and update sub-presenters for their respective data structures.

Homogeneous-Presenter is a sub-class of Structured-Presenter for structured objects whose components are all of the same type. The sub-classes of Homogeneous-Presenter can use a single sub-presenter to display all the components of an homogeneous structured object, thus saving a large amount of storage. Homogeneous-Array-Presenter and Homogeneous-List-Presenter are sub-classes of Homogeneous-Presenter specialized to arrays and lists.

Rectangular-Alignment-Mixin provides definitions to align sub-presenters in columns or rows.

String, Icon, Bitmap and Color-Presenter provide commonly used presentations.

The chess program uses many of these classes. For example, the piece-presenters are instances of Icon-Presenter, board-presenter is an instance of Homogeneous-Array-Presenter, and it uses the Borders-Mixin to define the borders of the board. The menu containing the save and quit commands is presented using a List-Presenter to present the list of save-command and quit-command, and uses the Rectangular-Alignment-Mixin

to align the presentations in a column, left aligned, and with some space between the items.

Nephew presenters, commands and recognizers are similar to Smalltalk's MVC views and controllers [Goldberg83]. The main difference is that the role of the controllers is played by commands and recognizers in Nephew. By separating gesture handling (recognizers) from dialogue control (commands), Nephew simplifies the design of the controllers.

Presenters, controllers and recognizers are also similar to MacApp's [Schmucker86] views and commands. The role of commands in both systems very similar, serving to collect the inputs for the program operations. Nephew takes the idea one step further by using commands as an object representation of the program operations, and allowing a command to be used anywhere where an object can be used. For example, the commands can be displayed with a presenter.

6 Classification and Separation In Intelligent Interfaces

This section first gives a definition of *intelligent interface* and then shows how communication concepts and Nephew could be used to construct interfaces that are intelligent according to the definition given below.

Intelligent Interfaces

When a program communicates with the user it has to make certain decisions, called *communication decisions*, about the concepts it communicates. The program must decide *what* information to communicate to the user, *when* to communicate it, and *how* to encode it. Also, the program must *decode* the input from the user, and then *interpret* the decoded concept.

A user interface can be called *intelligent* in the measure to which communication decisions are conditioned on an analysis of the information listed below [Rissland82]:

Program model: a model of the capabilities of the program, such as the objects it supports, and the operations it provides. Also included here is a description of the program's user interface design choices (*e.g.* whether the program uses a menu to display the operations).

User model: a model of the characteristics of the user of the program, such as his or her expertise, preferences and past history of interaction with the program.

Task model: a model of the tasks the user wants to accomplish when using the program.

Workstation model: a model of the characteristics of the workstation such as its operating system, speed, input and output devices.

Knowledge about interface design: knowledge about graphic design, wording of error messages, interaction styles, etc.

For instance, an interface that decides whether to use a menu or a command line interface based on an analysis of the user model (*e.g.* to find out how familiar a user is with a program) is more intelligent than one that makes the decision irrespective of the information contained in the user model.

There are two major ways in which these models can be used. The difference comes from whether the knowledge is used only at the time the program is designed, or whether the program itself can reason with this knowledge at run time.

When the knowledge is used only at design time, it is used to make user interface design decisions, which are then hardwired into the specification of the program. Software tools that aid in incorporating intelligence into a user interface at design time can be called *designer's assistants*.

When the knowledge is used at run time the program can make user interface design decisions tailored to the specific user and specific interaction problems that occur while the user is interacting with a program. These kind of interfaces can be called *adaptive*.

Figure 2 shows how interfaces are constructed using Nephew. The interface designer, represented by the box labeled **reasoning component** chooses presenters, commands and recognizers from the **building block library**, and glues them together using **Nephew**.

In Nephew's current implementation, the reasoning component is a human acting at design time rather than at run time. A designer selects building blocks from the Nephew library, tailors them to a specific application, and defines the connections between them. Given the architecture of Nephew, part of the reasoning component could be replaced by a designer's assistant and an adaptive interface. The designer's

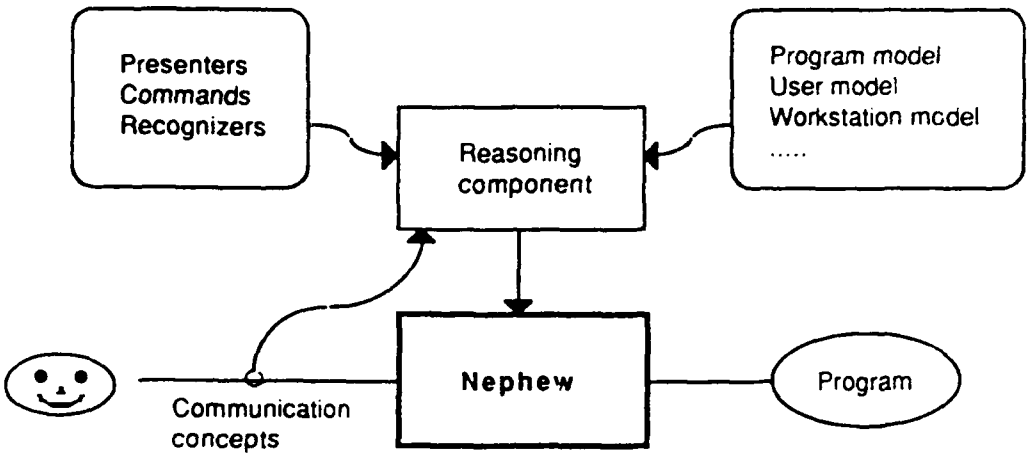


Figure 2: Constructing interfaces with Nephew.

assistant would help the human designer choose and tailor the building blocks. The adaptive interface would be part of the Nephew run time environment, and it could tailor and replace building blocks after reasoning about the state of the dialogue with the user. The following subsections discuss how communication concepts and the architecture of Nephew facilitates the automation of the reasoning component.

Revising Communication Decisions

Intelligent interfaces operate by generating or revising communication decisions. A designer's assistant generates communication decisions at design time, by suggesting different input techniques and different ways to display information. The human designer selects from the possibilities offered by the assistant, and then iterates revising the decisions before settling on a design. The usefulness of such an automated assistant is hampered if the different designs cannot be quickly implemented and tested. Separation allows replacing the modules that implement the different designs without reprogramming, so separation facilitates testing different designs. An adaptive interface places even more stringent requirements on separation because the design decisions are changed at run time, and hence must be made without reprogramming.

For instance, consider the following alternative interface designs for an operation with a single parameter.

- The parameter gets the value of the currently selected object, and the operation is chosen from a menu. The Macintosh interface has many examples of this design (*e.g.* cut and paste).

- The operation is presented as an icon next or close to the presentation of the object that it acts upon. Clicking on the operation icon invokes the operation. For instance, in the Macintosh interface the "close-window" operation is presented as an icon, called the *close box*, in the top left corner of the window to which it applies. Each window has its own close box.

- The operation is presented as an icon and the parameter is specified by dragging a presentation of the object into it. For example, the Macintosh Finder presents the "delete-file" operation as a trash-can icon. Files are deleted by dragging their presentation to the trash icon.

In Nephew all these interfaces are defined in terms of the set of communication concepts discussed in section 3. So, the modules that implement the different interface designs can be plugged into the functionality portion of the application without reprogramming. The revised communication decisions suggested by a designer's assistant could be implemented automatically and tested immediately by the human designer.

Reasoning About Building Blocks

A problem with many tool-kits is that it is hard to find the appropriate building block for a given situation. Classification can be used to construct a designer's assistant to help user interface designers to find the right building block and to tailor it for a given situation.

Communication concepts can be used to describe building blocks, providing a powerful language for designer's assistants to index into a database of building blocks. A query by example browsing tool such as Rabbit [Tou82] or Backbord [Yen88], can then be used to find candidate building blocks given the specification of a few attributes of the building block.

For instance, suppose the drag recognizer building block was described as follows:

> The dragger can be used to decode the **activate** communication concept.
> It is appropriate for operations that are presented as an icon.
> It is appropriate when the potential parameter objects are presented as icons.
> It can be used with operations that require at least one input.

Suppose the designer's problem is to design the input side of the interface for an operation with a single parameter. The designer first specifies a query for a recognizer to activate an operation with a single parameter. Given that many recognizers can activate operations, the browsing tool returns multiple choices (*e.g.* the designs for single parameter operations discussed above). The designer can then refine the query, say, by specifying that both the operation and the parameters will be displayed as icons, and narrow down the set of building blocks to find the above dragger.

Describing the building blocks in terms of the communication concepts also facilitates constructing consistent interfaces. For instance, if the designer specifies a command icon that presents the "command active" communication concept using reverse video, then the tool can easily detect an inconsistency if the designer specifies another command icon that also uses reverse video to present the "correct input" communication concept. The designer's assistant can point out the inconsistency and suggest ways to correct it.

7 Final Remarks

The language of communication concepts described in this paper has two salient features:

- The communication concepts specify, in abstract terms, *what* information a program communicates with a user, without specifying *how* that information is communicated.

- The communication concepts support the transmission of all the information needed to implement a wide variety of graphical interfaces.

The main consequence of these two features is that communication concepts can be used to define an application/user-interface interface, that is, the interface between the application and user interface modules of a program. The language specifies, not only *what* information must be supplied by the application module in order to support the user interface, but also *how* this information must be encoded.

This language is good from the modularity and code reusability point of views. The user interface can be changed without affecting the application portion of the

program, and the interface building blocks can be reused because they are plug-compatible with the application portion of the program. Such a clean architecture is important for the construction of intelligent interfaces because it facilitates revising user interface design decisions. Also, communication concepts are concepts that intelligent interfaces can reason about when making design decisions.

Acknowledgements I am grateful to the following people for their helpful comments: Richard Cohn, Mike DeBellis, John Granacki, Brian Harp, Phil Hayes, Richard Lerner, Brad Myers, Robert Neches, Barbara Staudt, Joseph Sullivan, Sherman Tyler and John Yen. I also want to thank Kim Chau Luu for her help with the figures.

References

[Arens88] Arens, Y., Miller, L., Shapiro, S. C., and Sondheimer, N. K. 1988. Automatic Construction of User-Interface Displays. In *AAAI 88, The Seventh National Conference on Artificial Intelligence*, pages 808–813.

[Encyclopae74] Encyclopaedia Britannica, Inc. 1974. *Micropaedia*, page 45. Volume 3, Encyclopaedia Britannica, 15 edition.

[Foley82] Foley, J. and van Dam, A. 1982. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley.

[Foley88] Foley, J., Gibbs, C., Kim, W. C., and Kovacevic, S. 1988. A Knowledge-Based User Interface Management System. In *CHI'88 Conference Proceedings*, pages 67–72, ACM.

[Goldberg83] Goldberg, A. and Robson, D. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass.

[Green86] Green, M. 1986. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244-275.

[Hayes85] Hayes, P., Szekely, P., and Lerner, R. 1985. Design Alternatives for User Interface Management Systems Based on the Experience with COUSIN. In *CHI'85 Conference Proceedings.*

[Kamran83] Kamran, A. 1983. *Issues Pertaining to the Design of a User Interface Management System*, pages 43–48. Springer-Verlag.

[Moran81] Moran, T. 1981. The command language grammar: a representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies*, 15:3–50.

[Newman79] Newman, W. and Sproull, R. 1979. *Principles of Interactive Computer Graphics.* McGraw-Hill.

[Olsen86] Olsen, D. R. J. 1986. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, 5(4):318–344.

[Rissland82] Rissland, E. 1982. Ingredients of Intelligent User Interfaces. *International Journal of Man-Machine Studies*, 21:377–388.

[Schmucker86] Schmucker, K. J. 1986. *Object-Oriented Programming for the Macintosh.* Hayden Book Company.

[Smith84] Smith, R., Lafue, G., and Vestal, S. 1984. Daclarative Task Description as a User-Interface Structuring Mechanism. *Computer*, 29–38.

[Szekely88] Szekely, P. 1988. *Separating the User Interface from the Functionality of Application Programs.* Ph.D. thesis CMU-CS-88-101, Carnegie-Mellon University.

[Tanner83] Tanner, P. and Buxton, W. 1983. Some Issues in Future User Interface Management System (UIMS) Development. *IFIP WG 5.2 Workshop on User Inteface Management.*

[Tou82] Tou, F. F., Williams, M., Fikes, R., Henderson, A., and Malone, T. 1982. RABBIT: An Intelligent Database Assistant. In *AAAI 82, The National Conference on Artificial Intelligence*, pages 314–318.

[Yen88] Yen, J., Neches, R., and DeBellis, M. 1988. Specification by Reformulation: A Paradigm for Building Integrated User Suppport Environments. In *AAAI 88. The Seventh National Conference on Artificial Intelligence*, pages 814–818.